PDE extension
Changes over Levon's extension

Jan Šilar
jan.silar@lf1.cuni.cz

June 6, 2014

New extension is compared to Levon's work ([2]), mostly chapter 4

Domains Geometry Definition

Originally
see [2] – 4.3.1.1 and 4.3.1.2

Saldamli defines domain shape by listing its boundaries. Individual boundaries (points in 1D, curves in 2D resp. surfaces in 3D) are describes by shape-functions. Shape-function maps intervals ([0,1] for curves, [0,1]×[0,1] for surfaces) onto the boundary.

Example circular domain:

```
class Circle
  extends Boundary(ndims=2);
  parameter Point c = {0,0};
  parameter Real r = 1;
  redeclare function shape
    input Real tau;          //tau in [0,1]
    output Real coord[2];
  algorithm
    coord := c + r * { cos(2*Pi*tau), sin(2*Pi*tau) };
  end shape;
end Circle;


type CircularDomain
  extends Cartesian2D(boundary = {circle});
  parameter Point center;
  parameter Real radius;
  parameter Circle circle (c = center, r = radius);
end CircularDomain;
```

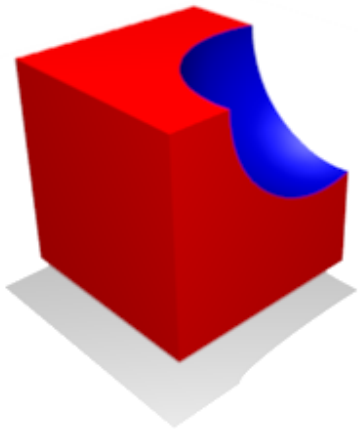

Figure 0.1: Boundary in 3D

Problem
This approach doesn't work well in 3D: if boundary is composed of several surfaces, parameters (arguments) of shape-function of these surfaces must be bounded not just in $[0,1] \times [0,1]$ interval but in some more complex set for each boundary surface so that they form a continuous boundary, e.g. see fig. 0.1. And there is no way to write this in Levon's extension.

Even if the syntax allowed this, it would be difficult for the user to compute these sets where parameters are bounded.

There is also no simple way to generate grid points during translation/solution.

Alternative approach
According to Peter's book [1] – 8.5.2, we define interior and boundaries of domain (these elements are called *regions* here) with one shape-function and for each region specify intervals for the shape-function arguments. This approach isn't more general (actually less), but is consistent in 1, 2 and 3D and (to me) is more natural. Inner points may be generated using this shape-function.

Modified `Domain` built-in type:

```
type Domain
  parameter Integer ndims;
  Real cartesian[ndims];
  Real coord[ndims] = cartesian;
  replaceable Region interior;
  replaceable function shape
    input Real u[ndims];
    output Real coord[ndims];
  end shape;
end Domain;
```

`Region` built-in type instead of `Boundary`:

```
type Region
  parameter Integer ndims;    //dim of space
  parameter Integer ndimr;    //dim of region
  parameter Real[ndims][2] interval;
  replaceable function shape;
    input Real u[ndims];
    output Real coord[ndims];
  end shape;
end Region;
```

Example circular domain:

```
class DomainCircular2D
  extends Domain;
  parameter Real radius;
  parameter Real[2] c = {0,0};
  function shapeFunc
    input Real r,v;
    output Real coord[2];
  algorithm
    coord := c + radius * r * { cos(2*Pi*v), sin(2*Pi*v) };
  end shapeFunc;
  Region2D interior(shape = shapeFunc, interval = {{0,1},{0,1}});
  Region1D boundary(shape = shapeFunc, interval = {1,{0,1}});
end DomainCircular2D;
```

Modified version
Here is equation instead of shape-function and new coordinate system instead of parameters of shape-function:

```
class DomainCircular2D
  extends Domain;
  parameter Real radius = 1;
  parameter Real c = {0,0};
  Real r, theta;
  Region2D interior(theta in (0,2*C.pi), r in (0,radius));
  Region1D boundary(theta in (0,2*C.pi), r = radius);
equation
  //coordinate transformation equation:
  coord = c + r * { cos(theta), sin(theta) };
end DomainCircular2D;
```

More complex geometries

More complex geometries may be defined using *Constructive Solid Geometry* – it is applying union, intersection and difference on previously defined shapes. The syntax is not designed already. It should be also possible to define domain in external file from some CAD app.

Differential operators

4.3.2

Partial derivatives

Originally
see 4.3.2.1
e.g. $\frac{\partial u}{\partial t}$ .. `der(u)`, $\frac{\partial^2 u}{\partial x \partial y}$ .. `der(u,x,y)`

Problem
There is no way to write mixed time and space derivative $\frac{\partial^2 u}{\partial x \partial t}$ in this notation.

This notation doesn't agree with mathematics, where we have different operators for ordinary ($\frac{du}{dx}$) and partial ($\frac{\partial u}{\partial x}$) derivatives.

`der(u)` for partial time derivative is confusing.

Alternative approach
$\frac{\partial u}{\partial t}$ .. `pder(u,time)`
$\frac{\partial^2 u}{\partial x \partial y}$ .. `pder(u,x,y)`
Now we can also write $\frac{\partial^2 u}{\partial x \partial t}$ as `pder(u,x,time)`

Normal derivative and normal vector

Originally
see 4.3.2.2
normal vector is implicit member of domain

Problem
Normal vector makes sense only in regions of dimension *n-1* in *n*-dimensional domain (i.e. surface in 3D, curve in 2D and point in 1D). There is no normal vector in *n* dimensional region and infinitely many in less than *n-1* dimensional regions.

Alternative approach
normal vector `n` is implicit member of all *n-1* dimensional regions in *n*-dimensional domain. So we write

```
pder(u,omega.boundary.n) = 0 in omega.boundary;
```

some shortcuts suggested later.
Using normal vector outside differential operators should be also possible e.g.:

```
field Real[3] flux;
flux*omega.boundary.n = 0 in omega.boundary;
```

Accessing coordinates and normal vector in der() operator

Originally
not discussed

Problem
Coordinates and normal vector are defined within the domain class, but they are used in equations that are written outside domains. Thus they should be accessed using `domainName.` prefix (e.g. `omega.x`), which is tedious.

In the example in 4.3.2.2 the normal vector `n` is reached outside the domain class without `domainName.` prefix even thou it is defined in the domain. It is not explained how this is enabled.

Solution
Fields are differentiated with respect to coordinates or normal vector only (or may be also some other vector for directional derivative??). Thus in place of second and following operands of `pder()` operator may be given only coordinates or normal vector. So variables in this positions may be treated specially and coordinates and normal vector of the domain of the field being differentiated may be accessed without the `domainName.` prefix here.

If coordinates or normal vector is used in different way (not in place of second and following operands of `pder()`), an alias for it may be defined in the model, e.g.

```
Real x = omega.x;
```

or `(discussed later)`

```
coordinate Real x = omega.x;
```

Start values of derivatives

Originally
not discussed

problem
Higher derivatives are allowed for fields thus we need to assign initial values to its derivatives sometimes.

solution
New attributes `startPrime` and `startSecond` (May be `startSecond` is not needed??) for field variables are introduced. Usage e.g.:

```
field Real u(start = 0, startPrime = field(sin(x*y) for (x,y) in omega);
```

Initial values for higher derivatives must be assigned in `initial equation` section.

Usage of `in` operator

PDEs

Originally
see 4.3.3

`in` operator is used for BCs to specifie on which region they hold. It is not used for PDEs.

problem
PDEs hold on particular regions (usualy `interior`) as well. Specification of region for BC but not for PDE is confusing. Besides that, the domain may be splited into separate regions and different PDE may hold on each region.

solution
We suggest to use `in` operator also in PDEs to specifie the region, as it is also in [1].

Accessing field values

Originally
see 4.2.4, in function-like style

problem
It is not consistent with curent Modelica – to access values of regular variables in particular time in this function-like style is also not allowed.

If more then one coordinate system are defined in a domain (discussed later), it is not clear which coordinates are used in the function-like expression.

solution
Regions consisting of one point and the `in` operator will be used instead to represent the particular point. E.g.

```
model heatPID
  record Room extends DomainBlock3D;
    Region0D sensorPosition(shape = shapeFunc, range = {{1, 1}, {0.5, 0.5}, {0.5,
0.5}});
  end Room
  Room room(...)
  field Real T(domain = room);
  Real Ts;
  ...
equation
  Ts = T in room.sensorPosition;
  ...
end heatPID;
```

`in` operator will be probably used also to match regions from different domains and to write equations (boundary conditions) relating fields from different domains. The syntax is not developed yet.

Modifications presented below are not so important and are questionable.

Coordinates

Originally
see 4.3.1.1 and 4.3.1.3

There are two arrays for coordinates predefined in the built-in `Domain` type. `cartesian` for cartesian coordinates and `coord` for arbitrary coordinates specified by the user. No other coordinates may be defined except aliases to elements of these predefined arrays.

Problem
May be this is not flexible enough. User may need more different coordinate systems.

Solution
new modifier `coordinate` to define coordinates. Usage e.g.

```
"coordinate Real" coordName;
```

The array `coord` in the built-in `Domain` type may be left out then.

Field literal constructor

originally
see 4.2.2, e.g.:

```
u = field(2*a+b for (a,b) in omega)
```

where iterator variables `(a,b)` exist only in constructor expression and represent coordinates in omega (probably `coord`, but may be `cartesian`, it is not clear from the document.)

problem
It allows to define the field values in terms of only one coordinate system. There may be two (or more – if `coordinate` keyword admitted) coordinate systems defined and it may be useful to be able to define fields using any of them.

And the syntax suggested below is shorter and simpler anyway.

solution
modified syntax:

```
"field (" expr "in" dom ")"
```

or just shortcut

```
"{" expr "in" dom "}"
```

where `dom` is a domain and `expr` may depend on coordinates defined in this domain. E.g.

```
field Real f = field(2*omega.x+omega.y in omega.interior);
```

Further problems

If more coordinates are defined, which are considered when defining a region? This problem is solved, if we use equations instead of shape-functions.

References

[1] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, 2004.

[2] Levon Saldamli. *A High-Level Language for Modeling with Partial Differential Equations.* PhD thesis, Department of Computer and Information Science, Linköping University, 2006.